NTNU

Norwegian University of
Science and Technology

Reliable and secure storage with Erasure Codes for OpenStack-Swift in PyECLib

Cheng Chang

Master in Security and Mobile Computing
Submission date: June 2016
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title:	Reliable and secure storage with erasure codes for OpenStack Swift in PyECLib
Student:	Cheng Chang

Problem description:

Erasure coding is a technology which is now widely used in storage systems in order to provide redundancy. Comparing to the traditional replication technique when considering fault-tolerance for the storage, erasure coding could provide higher levels of fault-tolerance with less cost on the storage space. The basic idea of erasure coding is to broke the data into several pieces. Each piece is coded in a scheme to contain a part of the original data, which makes it possible to recover the full data while there is loss of some pieces.

Erasure coding could effectively reduce the storage cost when designing storage systems, especially for cloud storage services such as Amazon S3 and Microsoft Azure. The motivation of this project is to compare the storage fault-tolerance mechanisms provided by erasure coding with the traditional approach of simple replication. In addition, the project will also look at the possibilities of encryption offered by Duplicity, which is an open source software used to provide encrypted backups. The implementations of components and benchmark testing are going to be performed on Swift, an object storage framework provided by the open IaaS cloud platform OpenStack.

The goals of the project are:

- Be familiar with OpenStack and Swift framework and build a benchmark testing environment.
- Test and compare erasure coding in different scenarios against the simple replication approach, both with or without the encryption mechanisms provided by Duplicity.
- Develop a plugin for Swift to implement an erasure coding scheme provided by NTNU, and test its performance comparing with several other erasure coding.

Assignment given:	15 January, 2016
Supervisor:	Danilo Gligoroski, ITEM, NTNU
	Johan Montelius, ICT, KTH

Abstract

In the last decade, cloud storage systems has experienced a rapid growth to account for an important part of cloud-based services. Among them, OpenStack Swift is a open source software to implement an object storage system. Meanwhile, storage providers are making great effort to ensure the quality of their services. One of the key factors of storage systems is the data durability.

Fault tolerance mechanisms play an important role in ensuring the data availability. Existing approaches like replication and RAID are used to protect data from lost, while with their own drawbacks. Erasure coding comes as a novel concept applied in the storage systems for the concern of data availability. Studies showed that it is able to provide fault tolerance with redundancies while reducing the capacity overhead, offering a tradeoff between performance and cost.

This project did an in-depth investigation on the OpenStack Swift and the erasure coding approach. Analysis on erasure coded and replication systems are performed to compare the features of both approaches. A prototype of custom erasure code is implemented as an extension to Swift, offering data storage with promising reliability and performance.

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) and Kungliga Tekniska högskolan (KTH) as the master thesis of my Master of Security and Mobile Computing.

I wish like to express my sincere gratitude to my supervisors, Professor Danilo Gligoroski at NTNU and Professor Johan Montelius at KTH, who helped me a lot throughout the project. The quality of my project greatly increased through their valuable feedback and support.

Finally, I would like to thank my friends and professors at both universities, wish them all the best.

Contents

List of Figures

List of Tables

List of Source Code Snippets

List of Abbreviations

API Application programming interface.

CPU Central Processing Unit.

CRS Cauchy Reed-Solomon.

DDF Disk Drive Format.

FEC Forward Error Correction.

FTP File Transfer Protocol.

GnuPG GNU Privacy Guard.

HDFS Hadoop File System.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

I/O Input/Output.

IaaS Infrastructure-as-a-Service.

ISA-L Intel Intelligent Storage Acceleration Library.

KTH Kungliga Tekniska högskolan.

LRC Local Reconstruction Code.

MDS Maximum Distance Separable.

NTNU Norwegian University of Science and Technology.

RAID	Redundant Array of Independent Disks.
REST	Representational State Transfer.
RS	Reed-Solomon.
SAIO	Swift All In One.
SCP	Secure copy.
SNIA	Storage Networking Industry Association.
SSH	Secure Shell.
URL	Uniform Resource Locator.
XOR	exclusive or.

Chapter 1
Introduction

1.1 Motivation

Cloud storage services have experienced a significant growth in recent years. Since a prototype of cloud storage service first appeared in 1980s, more and more companies are making efforts to bring products to either commercial or private users, such as Amazon S3, Microsoft Azure Storage, Google Drive and Dropbox. According to a recent market research report, the global cloud storage market is expected to grow from USD 18.87 billion in 2015 to USD 65.41 billion by 2020[1], at the annual rate of 28.2%.

Contents which are being stored in cloud storage services may vary, including private digital data such as photos and videos, as well as critical business related data, databases and backups. To deliver a satisfying cloud storage service, storage capacity or price are not the only things that users care about. Instead, ensuring good reliability and availability of data is really important to the clients.

To protect data from the system failures, simple approach of replication was applied widely. By using replication, copies of data are replicated and stored in different locations of hardware devices, which provides the protection when a single copy is lost or damaged. However, replication also brings extra cost for the additional storage space for the replicas. Another approach of Redundant Array of Independent Disks (RAID) was also introduced to save the overhead of storage space, while it does not scale so well when dealing with high rate of failures.

Erasure coding could provide protection against data lost, as well as reduce the storage overhead comparing to replication. The concept of erasure coding is to process the original data and generate several parity data fragments according to a certain scheme. Those parity data fragments could be further used to reconstruct the source data when needed. Meanwhile, it only requires a relatively lower overhead to store the parity data fragments. With such advantage, erasure coding has been applied in many large commercial and open source cloud storage systems, such as Microsoft Azure Storage[2], Facebook[3] and Google[4].

1.2 Scope and Goals

This thesis is motivated by studying the erasure coding used in cloud storage systems. The project intends to compare the fault tolerance mechanisms provided by erasure coding with the traditional approaches of simple replication and RAID. Performances of different erasure coding schemes will also be tested and analyzed on an open source platform called Swift from OpenStack.

The goals of the project are identified as below:

- Theoretically analysis the erasure coding, and compare it with other existing approaches to enhance data availability.
- Install the OpenStack Swift framework and build a benchmark testing environment.
- Test and compare erasure coding schemes in different scenarios, as well as against the simple replication.
- Extend OpenStack Swift to support an erasure coding scheme provided by NTNU, and perform necessary tests on such implementation.

1.3 Outline

Chapter 2 presents the relevant background information and important key concepts for the project. The chapter introduces existing approaches in terms of enhancing data availability and durability. The concept of erasure coding is explained, along with brief introduction of several well-known erasure codes.

Chapter 3 introduces OpenStack platform, and the object storage system Swift, as the focus of this project. This chapter describes the fundamental concepts and architecture of Swift. The integration of erasure coding technology in Swift is also introduced. The chapter ends with examples of installation, configuration and basic operations.

Chapter 4 performs comparison test on erasure coding against simple replication. Different schemes of both approaches are tested to compare the data durability as well as the performance. Analysis of the test results are discussed afterwards.

Chapter 5 presents a plugin for extending OpenStack Swift to support using a custom erasure coding scheme, and applying it in a backup storage system using

Duplicity. The design and implementation of the prototype is presented and analyzed as well.

Chapter 6 concludes the project. The achievements and limitations are discussed, with the future work plan of the project.

Chapter 2 Background

This chapter will firstly introduce and analyze the existing data protection approaches in Section 2.1. After that, the erasure coding is introduced as the focus of the project in Section 2.2. The concept and idea of design for erasure codes will be explained, and ending with a brief introduction of several common erasure codes.

2.1 Traditional Data Protection Approaches

2.1.1 Replication

To protect data from being lost, the simplest way is to use replication in the storage system to provide redundancy for the data. One or more copies of the original data will be generated and stored, which are called replicas. Whenever there is something wrong happens to the original data such as disk failures, the data could be restored from the replicas, as an additional tolerance to avoid data loss.

The replication process happens at the stage of data write. The original data is duplicated into two or more replicas, which are then sent to different locations. To ensure best data durability, replicas are usually stored in different hardware drives, hosts, or even different geographical locations. To avoid performance disturbance, caching is usually applied so that the server could rapidly response to the client after one replica is completed, while generating and transferring redundant replicas in the background. On the other hand, the restore process of replication is as straightforward as simply pick an available replica and serve the request, since each replica is exactly the same.

The overhead of replication is quite obvious. It requires 2x or more storage space of the original data, depending on the replica factor, which is how many replicas are stored.

Replication is widely used since it is quite simple to implement. It is suitable to be applied in cluster or distributed systems, with low sensitive of storage space. Moreover, there is no extra management cost for applying replication. Hadoop File System (HDFS) uses triple replication as default storage policy, with replica factor being three[5].

2.1.2 Redundant Array of Independent Disks (RAID)

The term ‘RAID’ is the short of ‘Redundant Array of Inexpensive Disks’, which was first invented in 1988[6]. At the beginning, the idea of RAID was to get high performance using parallel operations with multiple general disk drives. With the development of hardware disk drives, nowadays more effort has been put on applying RAID in order to achieve better reliability by configuring for redundancy.

Different configurations in RAID are used to achieve different goals, which are called levels. Each RAID level employs striping, mirroring or parity to get better performance or reliability. Below lists the brief introduction of the two basic RAID levels, RAID-0 and RAID-1, together with several other common levels. More RAID levels are standardized by the Storage Networking Industry Association (SNIA) in the Common RAID Disk Drive Format (DDF) standard[7].

- RAID-0 uses striping to split data evenly to two or more disks. Thus, the overall throughput could be improved with concurrency to n times higher than a single disk. However, there is no redundancy provided by RAID-0. Oppositely, the data durability is worse than single disk scenario, as a failure on any disk in the stripe will cause the unrecoverable data loss.

- RAID-1 applies mirroring among several disks. To the contrary of RAID-0, it provides redundancy as data is copied to several pieces and stored on each disk. The data is available as long as any one piece of data is survived. RAID-1 is quite similar to a disk-level replication in concept.

- RAID-5 acts as a tradeoff between the cost and data availability. It consists of a distributed parity block that could be used to restore original data when a single disk failure occurs. It can be considered as a balance between RAID-0 and RAID-1.

- RAID-6 enhanced data protection based on the mechanism used by RAID-5. Another parity block is added and working separately with the parity block implemented by RAID-5, thus, more redundancy is provided.

To meet variety requirements of storage systems, it is possible to apply nested RAID levels or even create customized configurations. However, the drawbacks of RAID are quite obvious as well. The cost of applying RAID is relatively high since usually several hardware disk drives are required. Besides, for the performance concern a hardware RAID controller is required to manage the read and write, which also adds the cost. Meanwhile, disk drives with large storage capacity may take days or even weeks to rebuild, which increases the chance of a second disk failure during rebuild.

2.2 Erasure Coding

2.2.1 Concepts

Erasure coding is a technology that been widely used to ensure data availability by using coding theory. The idea of erasure coding comes from Forward Error Correction (FEC) techniques, which intend to provide the ability to reconstruct missing data with some added redundant information. The theory is based on the use of error detection and correction codes, which have been well studied and widely used in many fields such as multi-cast transmission[8][9], sensor network[10] and online video streaming[11].

Luigi Rizzo introduced the basic concept of erasure codes used in data storage in [12]. Figure 2.1 shows the graphical encoding and decoding process of erasure coding. The encoder reads the one block of source data, and divided in to k fragments. In addition, another m fragments are generated according to the encoding algorithm, acting as the redundancy of the source data. The same process applies to the following blocks until the entire source data is encoded. Thus, $n = k + m$ fragments are encoded as outcome. The k fragments are called data fragments and the m fragments are called parity fragments, while the coding scheme can be represented using a tuple (k, m).

Likewise, the source data can be decode from the encoded data, as shown in Figure 2.1. Sufficient encoded fragments are passed to the decoder, and the reversed computations are performed on the parity fragments to rebuilt the missing data. With sufficient data and parity fragments provided, the entire source could be reconstructed successfully.

With such encoding technique, data redundancy is provided for enhanced data durability. Conceptually the erasure codes are able to tolerate the data loss of a certain amount of fragments. The fault tolerance ability of an erasure code is defined by Hamming distance. Any set of failures strictly less than the Hamming distance can be tolerated[13]. In other words, the source data could be reconstructed with a subset of the encoded data, which is the main advantage of erasure coding. As there are a large number of disks and nodes in a storage system, distributing fragments

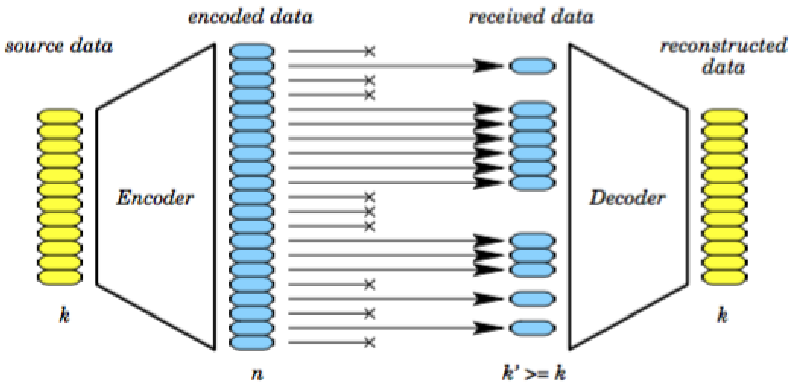

Figure 2.1: Process of encoding and decoding of erasure codes[12]

across the entire set of disks could effectively ensure the data availability in case of disk failures.

2.2.2 MDS and non-MDS codes

There are two types of erasure codes, with different designs in concept. The most common one is Maximum Distance Separable (MDS) codes. MDS codes can tolerate any m lost fragments using m redundant elements, thus it is optimally space-efficient[14]. The Hamming distance of a MDS code is equal to $m + 1$. Oppositely, non-MDS codes are non-optimal, as they are not able to tolerate all possible sets of m fragments lost. Several examples of both MDS and non-MDS codes are introduced in the following content.

Reed-Solomon (RS) Codes

Reed-Solomon codes are one of the most typical family of erasure codes[15], firstly being introduced in 1960[16], and widely used in the field of data transmission as well as storage systems. A Reed-Solomon code is MDS code, thus it delivers optimal fault tolerance with the space for redundancy. One of the most important feature of Reed-Solomon codes is that they can work with any k and m specified. The encoding algorithm of a Reed-Solomon code is based on XOR operations as well as multiply operations in a Galois filed GF(2^w)[17], provided as a n * k generator matrix. For example, there is always a Reed-Solomon defined with arithmetics in GF(2^8) for any storage system containing 256 disks or less[18]. Another example of a (4,2) Reed-Solomon code is demonstrated in Figure 2.2.

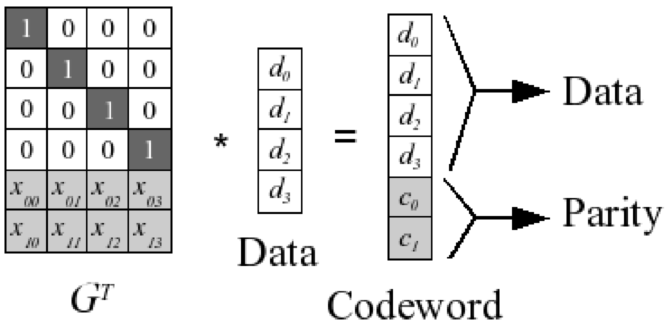

Figure 2.2: Coding process of a (4,2) Reed-Solomon code[19]

Cauchy Reed-Solomon (CRS) Codes

Comparing to regular Cauchy Reed-Solomon codes, Cauchy Reed-Solomon codes use Cauchy matrices instead of Vandermonde matrices. As every square sub-matrices of a Cauchy matrix is invertible, it can be transformed into multiple XOR operations[20] thus it reduces the computational cost with optimized implementation. More details about the theory of Cauchy Reed-Solomon codes are out of the scope of this project and could be found in [21].

Flat XOR codes

Flat XOR codes are one of the non-MDS codes. The Hamming distance of flat XOR codes is less than m, but certain failures at or over Hamming distance may also be tolerated. The encoding computations are based on simple exclusive or (XOR) operations instead of Galois Field arithmetic. Each parity fragment is simply the XOR of a subset of data fragments[14]. Meanwhile, the reconstruction process is more complicated, since there could more than one recovery equations for each data fragment. Using flat XOR codes could significantly reduce the computational cost comparing to the Reed-Solomon codes, especially with a dense generator matrix. However, such performance benefits come at the price of storage overhead.

2.2.3 Comparison with Replication and RAID

Erasure codes could be considered as a superset of simple replication and RAID techniques[22]. For example, the triple replication used in HDFS could be described as a (1,2) erasure code, and RAID-5 with four disk drives could also be described as a (3,1) erasure code.

However, there is a huge difference between erasure coding and RAID, as the former is per object level based instead of disk level based. Hence, erasure coding could provide more flexibility when configuring the policy.

Comparing to replication, erasure coding could offer the same level of fault tolerance with a relatively low space overhead. However, such benefits come at a price, the performance penalty. According to the theory, the parity fragments will be computed every time the data is written or updated. Extra computational cost is also required when decoding the data. Although there are performance overheads, applying erasure coding is suitable for large and rarely accessed data, for example, backup files. Early studies showed that it is computationally expensive to implement erasure codes[12], but the cost has been reduced with growing performance capacity and hardware based optimization solution (such as Intel Intelligent Storage Acceleration Library (ISA-L)).

Chapter 3 Introduction to OpenStack Swift

OpenStack Swift is the platform used in the project. The first two sections Section 3.1 and Section 3.2 of this chapter will introduce the components and infrastructures of both project. Section 3.6 will present how erasure coding works in Swift. Section 3.4 and Section 3.6 will demonstrate the configuration of Swift and usage of erasure codes in practical.

3.1 OpenStack

OpenStack[23] is a open source software platform for building and managing cloud computing, mostly deployed as an Infrastructure-as-a-Service (IaaS), as comparable to Amazon EC2. It was first launched in 2010 as a joint project between Rackspace Hosting and NASA, which intended to provide cloud computing services on top of general hardware platforms. Currently OpenStack is managed by a non-profitable corporate and benefits from active collaborators from the community.

OpenStack provides a variety of functionalities by a series of components running on top of the platform, such as Nova for compute, Neutron for networking, Swift for object storage, and Glance for image service. Each component provides a set of Application programming interfaces (APIs) to the users as the frontend, as well as for communicating with other components. Such architecture makes OpenStack highly extensible so that it is possible to make modifications while keeping the interoperability. All the service components work on top of a well-managed virtualization layer so that there is no need for the users to be aware of the underlying details.

With the great functionality, scalability and extensibility it brings, OpenStack attracts hundred of large companies to work with it and build their public or private cloud services, including PayPal, Yahoo and Intel.

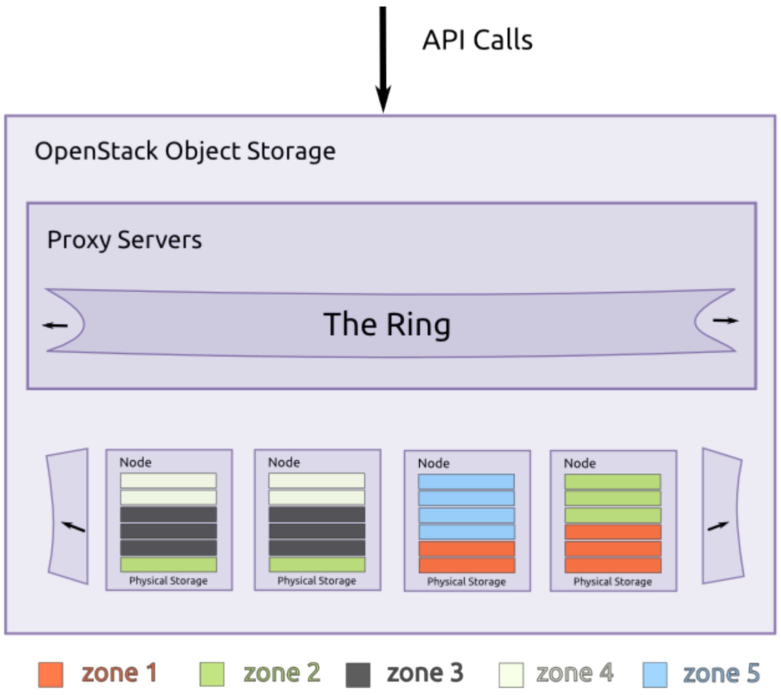

Figure 3.1: Architecture of Swift[25]

3.2 Swift

In this project, the focus is the OpenStack Object Storage component called Swift. It is designed to provide redundant and scalable distributed data storage using clusters of servers. The following descriptions in this section are partly summarized from the official documentation in [24] and [25].

As other components in OpenStack, Swift export a set of RESTful APIs to access the data objects. Operations like create, modify or delete can be done with standard Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS) calls. It is also possible to integrate the operations with language-specific APIs such as Python and Java, which are wrappers around Representational State Transfer (REST)-ful APIs.

Figure 3.1 illustrates an overview of the architecture of Swift. Some of the key components are introduced below.

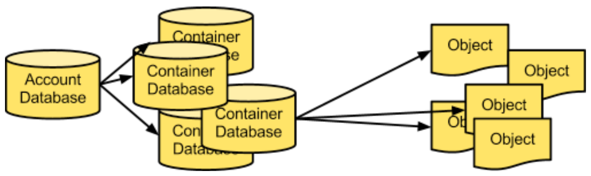

Figure 3.2: Hierarchy of entities in Swift[25]

3.2.1 Object Storage

As for object storage, the data stored is managed as individual objects[26]. Each object consists of not only the data, but also the metadata and an unique identifier. Object-based storage can be regarded as the convergence of file storage and block storage[27], which provides the advantages from both technologies: high level of abstraction as well as fast performance and good scalability.

Figure 3.2 shows that the Swift object storage system is organized in a three level hierarchy, which are Accounts, Containers and Objects from top to bottom, respectively. Objects are stored as binary files along with the metadata in the local disk. A container lists and maintains the references of all objects in that container, and a account lists and maintains the references of all containers. Each object is identified using a access path with the format as `/{account}/{container}/{object}`. Note that such hierarchical structure is not the same to how the objects are stored physically. Instead, the actual locations are maintained by the rings.

3.2.2 Rings

Since the data is distributed across the cluster, Swift uses the rings to determine the actual location of all the entities, including accounts, containers and objects. Conceptually, here is how the ring structures in Swift work: each ring has a list of the devices on all the nodes in the cluster. A number of partitions are created using the method of consistent hashing[28], and mapped to the devices in the list. Once an object needs to be stored or accessed, the hash value of the path to the object is computed with the same method, and then used as the index to indicate the corresponding partition. For the data availability concern, Swift support to isolate the partitions with the concepts of virtual regions and zones, so that replicas of entities will be assigned to different physical devices.

With the use of the ring structure, the storage system could have better scalability, especially for adding the storage capacity. It is possible to dynamically add or remove a node to the ring, without interrupting the whole service. Besides, since the structure is fully distributed, it helps to avoid the single point failure of centralized controllers in many other distributed file systems.

3.2.3 Proxy Server

The proxy server acts as the front-end of the Swift component and exposes the APIs to the users. It handles all the incoming requests, and looks up in the rings to locate the entities, then routes the requests to the entities transparently.

3.3 Erasure Coding in Swift

The support for erasure coding was added into Swift 2.3.0 since the Kilo release of OpenStack in 2015[29], as an alternative solution to replication for ensuring data durability.

Erasure coding support is implemented as a storage policy in Swift. Policies are used to specify the behaviors of the ring at the container level. There is one ring for each storage policy. As for erasure codes, the ring is used to determine the location of the encoded fragments. Such design allows different configurations such as replication counts or the allocation of devices, in order to meet different demands of performance and durability. Moreover, the storage policy is transparent from the application perspective, decoupling the underlying storage layer with the logistics in the applications.

There is no natively integrated support for encoding or decoding object data with erasure codes in Swift. Instead, Swift uses PyECLib as the backend library, which provides a well-defined Python interface. PyECLib supports a series of well-known erasure coding libraries, such as liberasurecode, Jerasure[30] and Intel ISA-L. Another good thing of PyECLib is the extensibility it provides so that it is possible to integrate custom erasure code implementations as a plugin.

The encoding and decoding of erasure codes are executed at the proxy server. For the write request, the proxy server continuously reads data from the HTTP packets and buffers into segments. PyECLib library is called to encode each segment into several fragments. The fragments are then sent to the locations specified by the ring, followed by fragments that are generated from the next segment. Similarly, when receiving a read request, the proxy server simultaneously requests the storage nodes for the encoded fragments. Note that not all the fragments are transferred, but only minimum k pieces of fragments which is specified by the erasure code scheme. The fragments are decoded by PyECLib to raw data and returned to the client.

The reconstruction occurs at different stages, which is executed at lower level on storage nodes of object servers, instead of the proxy server. The data stored is managed by the auditor as it routinely checks the fragments and metadata. Whenever an error is found (such as broken disks, flipped bit in data or mis-placed object), the auditor would call the object reconstructor to try to restore the missing fragments.

Other than acquiring all the fragments left, the reconstructor starts at asking the neighbor nodes for the fragments that are needed. As soon as enough fragments are gathered, PyECLib would be called to reconstruct the missing fragments.

3.4 Installation of Swift

As discussed in Section 3.6, the erasure coding support in Swift replies on PyECLib and a set of third party libraries, so first thing to do is to install these dependencies. Below is a list of the dependencies of Swift:

- **liberasurecode**[1] is an Erasure Code API library written in C with pluggable Erasure Code backends.
- **Jerasure**[2] is a library in C that supports erasure coding in storage application.
- **GF-Complete**[3] is a comprehensive library for Galois Field Arithmetic.
- **ISA-L**[4] is a erasure coding backend library written in ASM from Intel. This library comes with optimization for operation acceleration on Intel instruction sets including AES-NI, SSE, AVX and AVX2[31].
- **PyECLib**[5] provides Python interface for erasure coding support, based on liberasurecode.

After completion of installing the dependencies, a well-defined document of Swift All In One (SAIO)[32] provides a tutorial of installation of Swift development environment and setups to emulate a four node Swift cluster. Note that in order to setup an environment on single host, loopback devices should be set for storage.

3.5 Tutorial of basic Swift operations

The following tutorial is based on the SAIO environment setuped in Section 3.4.

The first step is to check whether Swift is running normally, and get `X-Storage-Url` as the public Uniform Resource Locator (URL) to access the object storage, and `X-Auth-Token` used for authentication. This can be done with command:

[1]liberasurecode: https://bitbucket.org/tsg-/liberasurecode/
[2]Jerasure: http://jerasure.org/
[3]GF-Complete: http://jerasure.org/
[4]ISA-L: https://01.org/zh/intel%C2%AE-storage-acceleration-library-open-source-version
[5]PyECLib: https://pypi.python.org/pypi/PyECLib

```
 $ curl -v -H 'X-Storage-User: test:tester' -H 'X-Storage-Pass:
↪  testing' http://127.0.0.1:8080/auth/v1.0
```

The authentication system in Swift will handle the request, validate the account and password pair in the header, and grant the access to the service. Figure 3.3 is example of the output from the server. The `X-Storage-Url` and `X-Auth-Token` obtained can be used in further interactions with Swift.

```
swift@swift-VirtualBox:~/bin$ curl -v -H 'X-Storage-User: test:tester' -H
 ↪  'X-Storage-Pass: testing' http://127.0.0.1:8080/auth/v1.0
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /auth/v1.0 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 127.0.0.1:8080
> Accept: */*
> X-Storage-User: test:tester
> X-Storage-Pass: testing
>
< HTTP/1.1 200 OK
< X-Storage-Url: http://127.0.0.1:8080/v1/AUTH_test
< X-Auth-Token-Expires: 86378
< X-Auth-Token: AUTH_tk38696483cfdd401084be4b9563d5aea3
< Content-Type: text/html; charset=UTF-8
< X-Storage-Token: AUTH_tk38696483cfdd401084be4b9563d5aea3
< Content-Length: 0
< X-Trans-Id: tx24b3541db8024074accfb-005743fda4
< Date: Tue, 24 May 2016 07:07:16 GMT
<
* Connection #0 to host 127.0.0.1 left intact
```

Figure 3.3: Example output of granted token

The example above shows the way to call the standard REST-ful Swift APIs using `cURL`. Alternatively, there is also a command-line interface to the OpenStack Swift API provided by `python-swiftclient\cite{swiftclient}`. For example, the statistics of an account could be displayed with command:

```
 $ swift -A http://127.0.0.1:8080/auth/v1.0 -U test:tester -K
↪  testing stat -v
```

Figure 3.4 shows the output statistics of the command, as `-A` denotes for the authentication URL, `-U` denotes for the username, `-K` denotes for the key.

```
swift@swift-VirtualBox:~/bin$ swift -A http://127.0.0.1:8080/auth/v1.0 -U
 ↪  test:tester -K testing stat -v
     StorageURL: http://127.0.0.1:8080/v1/AUTH_test
     Auth Token: AUTH_tk38696483cfdd401084be4b9563d5aea3
        Account: AUTH_test
     Containers: 0
        Objects: 0
          Bytes: 0
X-Put-Timestamp: 1464075341.07031
    X-Timestamp: 1464075341.07031
     X-Trans-Id: txde129cb60969435a95c9a-005744044d
   Content-Type: text/plain; charset=utf-8
```

Figure 3.4: Example output of `swift` command for statistics

3.6 Tutorial of using Erasure Coding in Swift

As introduced in section 3.6, erasure coding support is enabled in Swift as a storage policy. The storage policies are defined in `/etc/swift/swift.conf`. Figure 3.5 showes a sample configuration file of storage policies after configured as documented in section 3.4.

```
[swift-hash]
# random unique strings that can never change (DO NOT LOSE)
# Use only printable chars (python -c "import string; print(string.printable)")
swift_hash_path_prefix = changeme
swift_hash_path_suffix = changeme

[storage-policy:0]
name = gold
policy_type = replication
default = yes

[storage-policy:1]
name = silver
policy_type = replication

[storage-policy:2]
name = ec42
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 4
ec_num_parity_fragments = 2
```

Figure 3.5: Sample configuration of storage polices

As shown in Figure 3.5, the third policy defines an erasure code policy. It is configured to use the Vandermonde Reed-Solomon encoding implemented by `liberasurecode` library. The coding scheme is also defined as $k = 4$ and $m = 2$.

Next step to do is to create the corresponding object ring for the erasure code policy. The template of the ring create command with `swift-ring-builder` is:

```
$ swift-ring-builder <builder_file> create <part_power> <replicas>
↪ <min_part_hours>
```

Note that in order to apply erasure coding policy, the value of `replicas` should be set to the sum of `ec_num_data_fragments` and `ec_num_parity_fragments` in the policy, which is 6 in this case. Therefore, the command used to create a object ring for the 'ec42' policy (with the index of 2) is:

```
$ swift-ring-builder object-2.builder create 10 6 1
```

After the ring is built successfully, the devices should be added into the ring. Here the loopback devices are used, as listed in Figure 3.6. The `rebalance` command should be executed after adding the devices, so as to initialize the ring.

```
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6010/sdb1 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6010/sdb5 1
swift-ring-builder object-2.builder add r1z2-127.0.0.1:6020/sdb2 1
swift-ring-builder object-2.builder add r1z2-127.0.0.1:6020/sdb6 1
swift-ring-builder object-2.builder add r1z3-127.0.0.1:6030/sdb3 1
swift-ring-builder object-2.builder add r1z3-127.0.0.1:6030/sdb7 1
swift-ring-builder object-2.builder add r1z4-127.0.0.1:6040/sdb4 1
swift-ring-builder object-2.builder add r1z4-127.0.0.1:6040/sdb8 1
swift-ring-builder object-2.builder rebalance
```

Figure 3.6: Commands to add loopback devices to the ring

As previously introduced, the storage policy applies at the container level. So as to test with the erasure code policy, next step is to create a corresponding container for it. The storage policy of the container could be determined in the HTTP header, as specified in 'X-Storage-Policy' filed. The following commands show the process of creating a container using the 'ec42' policy and then upload a 1MB randomly generated test file into it:

```
 $ curl -v -X PUT -H 'X-Auth-Token:
↪  AUTH_tk6ad03f8b33a3427190514f751c24801e' -H "X-Storage-Policy:
↪  ec42" http://127.0.0.1:8080/v1/AUTH_test/ec_container
 $ dd if=/dev/urandom of=test.tmp bs=1k count=1000
 $ curl -v -X PUT  -T test.tmp -H 'X-Auth-Token:
↪  AUTH_tk6ad03f8b33a3427190514f751c24801e'
↪  http://127.0.0.1:8080/v1/AUTH_test/ec_container/
```

`swift-get-nodes` command helps to read the object ring file and display the underlying detailed information of the stored objects. As shown in Figure 3.7, the encoded fragments are placed at six devices, out while `sdb5` and `sdb8` are idle.

According to the theory, the (4, 2) Reed-Solomon code in use suppose to be able to tolerate any two missing fragments. To verify the data durability of erasure codes, two loopback devices folders are manually deleted to simulate the case of a two disks failures. Afterwards, the test file has been downloaded from Swift. The result of `diff` command shows that the original data has been restored successfully. The process of the verification is demonstrated in Figure 3.8.

```
swift@swift-VirtualBox:~$ swift-get-nodes /etc/swift/object-2.ring.gz AUTH_test
 ↪  ec_container test.tmp

Account          AUTH_test
Container        ec_container
Object           test.tmp


Partition        930
Hash             e893ebb4364585e15e9cfa0dbb986af0

Server:Port Device        127.0.0.1:6030 sdb7
Server:Port Device        127.0.0.1:6020 sdb6
Server:Port Device        127.0.0.1:6040 sdb4
Server:Port Device        127.0.0.1:6010 sdb1
Server:Port Device        127.0.0.1:6030 sdb3
Server:Port Device        127.0.0.1:6020 sdb2
Server:Port Device        127.0.0.1:6040 sdb8         [Handoff]
Server:Port Device        127.0.0.1:6010 sdb5         [Handoff]


curl -I -XHEAD "http://127.0.0.1:6030/sdb7/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2"
curl -I -XHEAD "http://127.0.0.1:6020/sdb6/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2"
curl -I -XHEAD "http://127.0.0.1:6040/sdb4/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2"
curl -I -XHEAD "http://127.0.0.1:6010/sdb1/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2"
curl -I -XHEAD "http://127.0.0.1:6030/sdb3/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2"
curl -I -XHEAD "http://127.0.0.1:6020/sdb2/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2"
curl -I -XHEAD "http://127.0.0.1:6040/sdb8/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2" # [Handoff]
curl -I -XHEAD "http://127.0.0.1:6010/sdb5/930/AUTH_test/ec_container/test.tmp" -H
 ↪  "X-Backend-Storage-Policy-Index: 2" # [Handoff]
```

Figure 3.7: Example of detail information of an object (partly)

```
swift@swift-VirtualBox:~$ rm -rf /srv/3/node/sdb3/ /srv/2/node/sdb6/
swift@swift-VirtualBox:~$ mv test.tmp test.original
swift@swift-VirtualBox:~$ swift -A http://127.0.0.1:8080/auth/v1.0 -U test:tester -K
 ↪  testing download ec_container test.tmp
test.tmp [auth 0.028s, headers 0.060s, total 0.065s, 28.342 MB/s]
swift@swift-VirtualBox:~$ diff test.original test.tmp && echo Same || echo Different
Same
```

Figure 3.8: Test process to verify restore file from subset of fragments

Chapter 4

Analysis on Erasure Coding comparing to Replication

As discussed in Chapter 2, erasure coding is considered to be an alternative approach to provide data protection and ensure data availability, as an addition to replication and RAID. The ideas behind erasure coding and RAID are quite similar in concept, as RAID acts on the disk level. On the other hand, replication is equal to a erasure code with the number of data fragments set to one, and the same number of parity fragments as replica factor. To find out the differences between replication and erasure coding, a system comparison will be performed on the common erasure codes used in practice with the replication approach. In particular, Reed-Solomon codes are analyzed in this comparison, since they are the most widely-used erasure coding families in practice.

This chapter consists of the analysis for data availability, space overhead and performance in Section 4.1, Section 4.2 and Section 4.3, respectively. Section 4.4 will conclude and discuss the results.

4.1 Availability

There are two situations that may happen affecting the data availability: the permanent loss of data, and the temporary unavailable. Temporary unavailable data may be caused by several factors such as broken links or crashes on servers. In this thesis, data unavailability refers to the cases that data is completely lost and unaccessible. Let p be the overall availability of the data, and α be the average availability of a single disk. To simplify the analysis, we assume the availability of each disk is independent from another.

4.1.1 Replication

In replication systems, the data availability is ensured by keeping several copies of the original data on different disks. Let k be the replica factor, which is the number of the copies. Therefore, the overall availability with a given replica factor k could be calculated as:

$$p = 1 - (1-\alpha)^k \tag{4.1}$$

4.1.2 Erasure Coding

According to the concept of erasure coding, a (k, m) erasure coded system divides the original data into k data fragments. In addition, m parity fragments are erasure encoded with the data fragments. The total number of fragments encoded is denoted by $n = k + m$. With such configuration, the system should be able to tolerate data loss less than m fragments. In other words, at least m fragments are required to ensure the data to be available from being reconstructed. Assume that all the encoded fragments are stored independently across all disks. Therefore, the overall availability is equal to the sum of probability of disk failures less than m, which is given as:

$$\begin{aligned} p &= \binom{n}{0}\alpha^n \quad + \quad \binom{n}{1}\alpha^{n-1}(1-\alpha) \quad + \quad \cdots \quad + \quad \binom{n}{m}\alpha^{n-m}(1-\alpha)^m \\ &= \sum_{i=0}^{m}\binom{n}{i}\alpha^{n-i}(1-\alpha)^i \end{aligned} \tag{4.2}$$

4.2 Space Overhead

The space overhead of replication systems is quite obvious, which is determined by the replica factor k. A storage system keeping k replicas of data has $(k-1) * 100\%$ overhead. For example, HDFS has space overhead of 200%, with the default triple replication scheme in use. Thus, the space efficiency of 3x replication is 33%.

In erasure coding storage systems, the space overhead is caused by the extra storage space consumed by the parity fragments. A (k, m) erasure code has space overhead of $\frac{m}{k} * 100\%$. For example, a Reed-Solomon code with 6 data fragments and 3 parity fragments (RS-(6,3) in short) has the space overhead of 50% (alternatively 66.67% space efficiency).

It is worthing that, although increasing number of parity fragments in erasure coding system do provide better data availability, however, it comes at a cost. Consider two erasure codes, RS-(6,3) and RS-(12,6). The space overhead of both systems are the same, 50%. However, the second system requires more disks to be involved in the system, which increases the probability of disk failures. Another consequence might be the increased latency. The encoded fragments are distributed across different disks, usually in different clusters or even different geographically locations. Since numbers of fragments are required to decode the data, the increasing cost on the communication over the links would cause the increasing latency of the service.

Table 4.1 shows the corresponding data durability and storage efficiency of several data protection schemes, assuming each single disk in use has the average availability of 0.995. The configurations of replication systems and erasure coded systems are picked as they are widely used in practice. For example, HDFS uses 3x replication by default, and Google uses RS-(6,3) in their clusters. As shown in the table, the result show that, with the same data availability provided, erasure coding approaches are able to cut the space overhead to roughly half of the one provided by replication approaches.

Policy	**Maximum Disk Failures Tolerated**	**Data Availability**	**Number of Nines**	**Space Overhead**
flat storage	0	0.995	2	0
2x replication	1	0.999975	4	100%
3x replication	2	0.999999875	6	200%
4x replication	3	0.9999999994	9	300%
RS-(4,2)	2	0.999997528	5	50%
RS-(6,3)	3	0.9999999228	7	50%
RS-(10,2)	2	0.9999734134	4	20%

Table 4.1: Comparison of availability and space overhead on schemes of replication and erasure codes

4.3 Performance

In this section, two sets of experiments are completed to compare the performance of replication and erasure coding systems. The experiments are designed to evaluate the performance with read operations and write operations.

4.3.1 Test Setup

The tests are performed in OpenStack Swift, which is mentioned in chapter 3. A single node Swift All In One virtual machine running on a Macbook Pro was used as the platform. Detailed performance specifications of the virtual machine are listed in Table 4.2 as the baseline.

CPU Cores	2
Memory Size	4096 MB
Operating System	Ubuntu (64bit)
memcpy **Speed**[1]	6739.96 MB/s
Disk I/O - Write Speed[2]	329 MB/s
Disk I/O Speed[3]	864 MB/s

Table 4.2: Test platform performance specification

The open source benchmarking tool *ssbench* (SwiftStack Benchmark Suite)[33] is used in the performance test. It is designed as a benchmarking tool for the OpenStack Swift with flexibility and scalability. In *ssbench*, there is a concept of $ssbench - worker$, acting as individual process to perform the requests to the Swift system. There could be one or more $ssbench - worker$ running either on the same host or across several client servers in the cluster. All the $ssbench - worker$ processes are using sockets to cooperate with the others and be managed by the $ssbench - master$ process. In this experiment, as the platform is a single-node virtual machine, all the tests are performed using one $ssbench - worker$. The version of *ssbench* suite in use is 0.3.9.

The tests are designed to simulate sending requests to Swift with a bunch of files. As files of different sizes may affect the performance, four categories of file sizes are involved in the test:

1. Tiny: files of size 4KB.
2. Small: files of size 128KB.
3. Middle: files of size 4MB.
4. Large: files of size 128MB.

The files used for the tests are generated by *ssbench*, which are defined in the scenario files. For each category of file sizes, two scenarios are defined to perform tests for both PUT and GET operations. The $crud_profile$ option in the scenario defines the distribution of each type of operation in the test, which are create, read, update and delete. Figure 4.1 shows an example of scenario file, describing a test that send PUT requests with 4KB files.

[1]measured with command: `mbw -b 4096 32 | grep AVG`

[2]measured with command: `dd if=/dev/zero bs=1024 count=1000000 of=test_1GB.data`

[3]measured with command: `dd if=test_1GB.data of=/dev/null bs=1024`

```
{
  "name": "4K write-only",
  "sizes": [{
    "name": "tiny",
    "size_min": 4000,
    "size_max": 4000
  }],
  "initial_files": {
    "tiny": 10
  },
  "crud_profile": [100, 0, 0, 0],
  "container_base": "ssbench",
  "container_count": 100
}
```

Figure 4.1: Example of scenario: 4KB files write

To compare the performance of replication and erasure coding, schemes of both approaches are picked with the same reason described in section 4.2 (except for the flat storage). The relevant policies of each scheme are defined in Swift, with the corresponding object rings created. In particular, $liberasurecode_r s_v and$ is chose for the implementation of the erasure codes, as it is the default option for erasure codes used in Swift. Tests of each policy are performed at the same duration of 30 seconds. The metrics of latency, throughput and operation performed within the period are recorded for analysis. The command used for test is listed below:

```
ssbench-master run-scenario -f <test_scenario_file> -u 1 -r 30
↪  --pctile 50 -A http://127.0.0.1:8080/auth/v1.0 -U test:tester -K
↪  testing --policy <policy_name>
```

4.3.2 Results

Figure 4.2 shows the average first byte latency of read operations on both systems, which could be considered as the measurement of the response speed of the system. The result shows that in general replication systems has much better read performance than the erasure coded systems. The main reason is that when serving the read requests, the proxy server of Swift needs to fetch enough data fragments and decode into raw data, which is pretty time consuming. On the contrary, the replication system just needs to hand one of the stored replica from the storage node to the client, without any extra effort. As shown in the graph, the latencies of the erasure coded systems grow with the size of the files, while there is no significant change in replication systems.

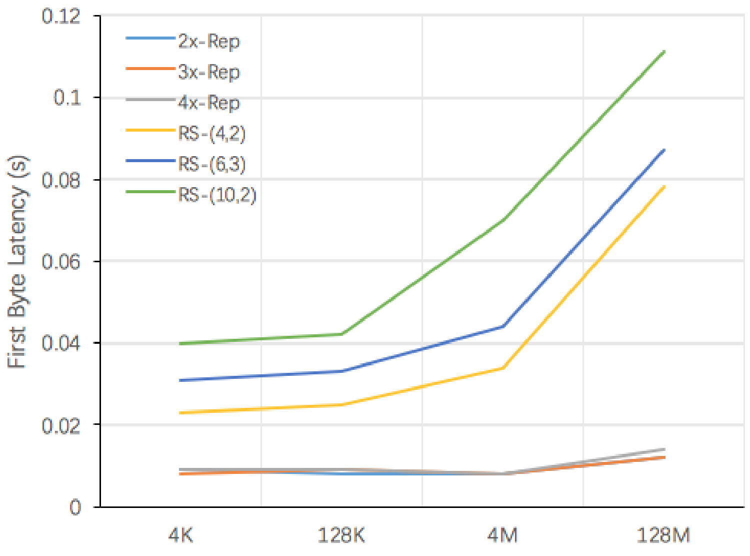

Figure 4.2: Average first byte latency of read operations of replication and erasure codes

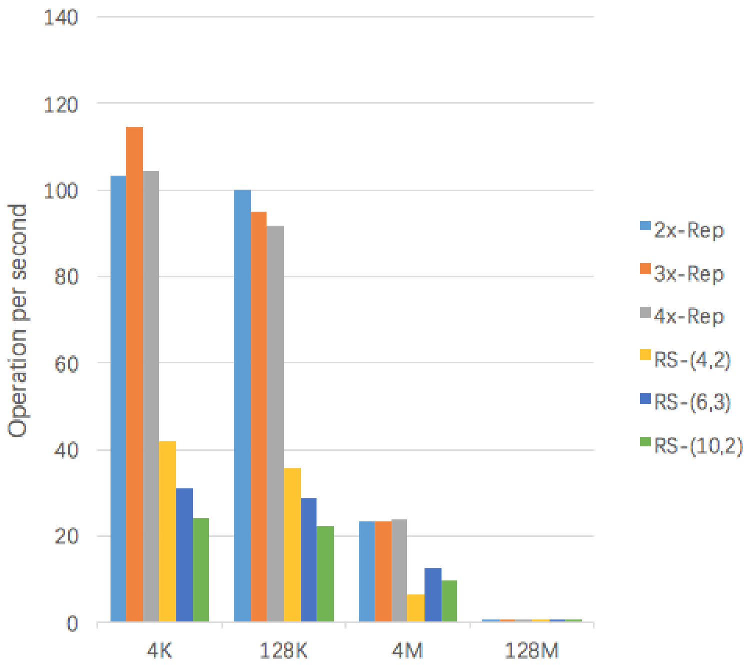

Figure 4.3: Average read operations completed per second of replication and erasure codes

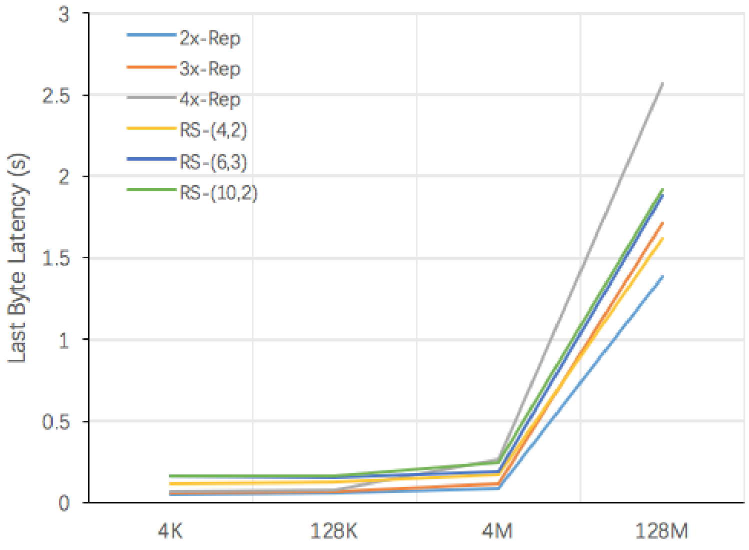

Figure 4.4: Average last byte latency of write operations of replication and erasure codes

Figure 4.3 shows the average read operations completed per second in both systems. With the growth of the file size, the values of replication systems drops from slightly over 100 operations per second for 4KB files, to 0.7 for 128MB files. Note that the replication systems completed around 23 operations per second on 4MB files, which is around 32 times of the operations on 128MB files, while the difference of file sizes is also 32x. The reason might be it achieved the system performance bottleneck. On the other hand, the operations of erasure codes were getting close to the replication as the growth of file size, which means the computation overhead of decoding became relatively smaller when comparing to the disk I/O.

Figure 4.4 shows the performance of write operations. The metric of last byte latency is recorded, which is equal to the total time used of a write operation. The results show that, the overall performance of replication system is slightly better than erasure coding. However, for 4MB and 128MB files, the three erasure codes outperformed the 4x replication. The trend of the reducing gap illustrates that erasure coded systems are good at writing large files. As for replications, several copies of original data are required to be written on the disks, which brings extra I/O overhead. The bigger size the file has, the more time would be consumed. While the single host environment might affect the results, same impact applies on distributed systems with the restriction of network bandwidth when dealing with large files and high replica factors. On the other hand, the extra parity data of erasure codes is relatively small, as the storage overhead is usually less than 100% as explained in

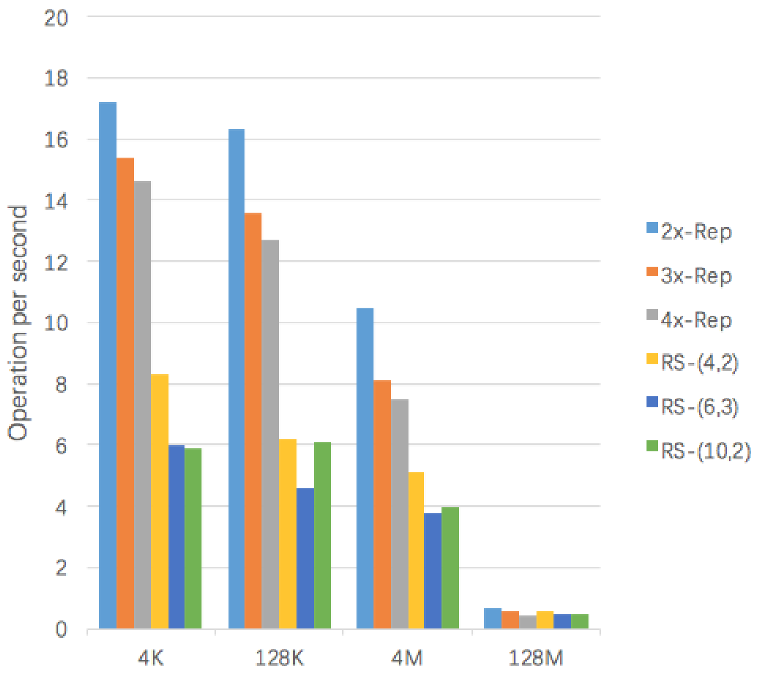

Figure 4.5: Average write operations completed per second of replication and erasure codes

section 4.2.

Figure 4.3 shows the average write operations completed per second. Similar pattern could be found as the read operations: the replications do better on small files, but erasure codes could achieve close or even the same performance on large files.

4.4 Discussion

As shown in the results, the erasure coding could be considered as an alternative approach to the replication to be used in some scenarios, although not a substitute.

The main benefits that erasure coding brings is the data durability. The theoretical analysis for the data availability provided by both erasure coding and replication showed that, the erasure coding approach could provide sufficient data durability as replication, while significantly reduce the required storage space for replicas. This feature can be quite important as it can save the investment on hard drives and reduce the management cost as well.

There is certainly shortcoming of erasure coding, which is the computational cost. Additional resources are consumed to encode and decode data, which affects the bandwidth and memory usage. However, the results of the tests showed the trend of reducing performance difference between erasure coding and replication, with the growth of the size of files. Therefore, erasure coding may achieve almost the same performance as replication, in the cases of writing huge data.

In conclusion, the results showed that erasure coding is good at processing huge data. The reduced storage overhead is an important feature, and the performance cost would be affordable if the data is not being read frequently. Thus, it might be a good idea to apply erasure coding on the storage of huge and hardly-updated data, such as system backups and digital images.

Chapter 5 Backup Storage with Swift

With the growing for the digital market, great importance has been given to the data itself. Digital data could play an important role in private life, such as photos and videos, or business field like commercial database, or even military area. It would be a irreversible disaster if data is lost. To avoid this, backup is used as a double insurance to the original data. The backups of data could be stored locally like Time Machine provided by Apple, usually at different hardware disks or devices. In recent decades, more and more backups are being stored in cloud storage providers such as Dropbox. Regardless of where the data is stored, the key idea is to keep the backups properly, and ensure the availability when they are needed.

In this chapter a prototype of backup storage system is demonstrated. The system utilizes the object storage provided by OpenStack Swift. The goal of the system is to ensure the data availability, while keeping the extra cost and overhead as low as possible. Two key features of the backup storage, data reliability and security, are the focus in design and implementation. To achieve good availability, a custom erasure code is implemented for the use in the prototype.

Section 5.1 presents Duplicity software for the use of generating the backups. The design and implementation of the custom erasure code is explained in Section 5.2. Section 5.3 demonstrates the usage of the prototype, and Section 5.4 presents the analysis of the system.

5.1 Duplicity

Duplicity[34] is an open source software suite which is able to create backups for the data. One advantage of Duplicity is that it supports incremental backup scheme. It generates a backup in the first place, and instead of updating the whole backup, Duplicity is able to update only the changes of the original data since last backup. Such ability is achieved by using `rsync` algorithm[35], which is a tool for detecting and synchronizing the changes of files and directories. As Duplicity only transfers part of data for each backup, it could save a lot of cost on time and bandwidth, and most importantly, storage space. Duplicity supports to transfer backups to a variety of destinations, including remote hosts using SSH/SCP, FTP servers, and commercial cloud storage providers like Amazon S3, Dropbox, and OpenStack Swift as repository as well.

Another good feature of Duplicity is that it supports encryption of the backups to ensure the confidentiality and integrity of the data, which is utilized in this system. Duplicity uses GNU Privacy Guard (GnuPG)[36] to perform encryption on data, which implements the OpenPGP standard as defined by RFC4880[37]. More details of Duplicity and its encryption mechanism can be found in Håkon's thesis[38].

Duplicity supports both symmetric and asymmetric encryptions, with a variety of algorithms. Asymmetric encryption do provide more protection. However, the encrypted backups can never be restored if the private key is lost, which is pretty likely to happen in a system failure. Thus, the symmetric encryption is chosen in the system.

5.2 Erasure Code

As previously discussed in the thesis, the erasure coding mechanism is able to provide good data availability while reducing the storage space overhead. Moreover, the performance cost of erasure codes is relatively close to the replication when processing large data. Since normally backups require high data availability and consist of hundreds of gigabytes or even terabytes of data, such features make erasure coding an ideal choice for the backup storage to ensure data durability.

5.2.1 Design

The erasure code used in the system is provided by Danilo Gligoroski, the supervisor of this thesis project. It is a XOR based code with (4,4) scheme (denoted by XOR-(4,4) in the following context). The four parity fragments with redundant information (denoted by y_n in the following context) could be calculated with XOR operations from corresponding subset of the data fragments (denoted by x_n in the following context). The equations of the encoding of parity fragments are listed in Table 5.1:

y_1	$= x_1 \oplus x_3$
y_2	$= x_2 \oplus x_4$
y_3	$= x_1 \oplus x_2 \oplus x_3$
y_4	$= x_2 \oplus x_3 \oplus x_4$

Table 5.1: Equations of encoding a custom XOR-(4,4) erasure code

The decoding process of the XOR-(4,4) code is different from the Reed-Solomon code previously introduced. As the XOR codes is not optimal codes, for one data fragment there could be one or more decoding equations, with different combinations of other data fragments and parity fragments. With the decoding equations listed in Table 5.2, data fragments x_1 and x_4 could be restored with two equations, while x_2 and x_3 have two more restoration equations. With larger k value selected for the XOR codes, more complicated decoding equations may appear, consisting of more elements.

x_1	$= x_3 \oplus y_1$
	$= x_2 \oplus x_3 \oplus y_3$
x_2	$= x_4 \oplus y_2$
	$= x_1 \oplus x_3 \oplus y_3$
	$= x_3 \oplus x_4 \oplus y_4$
	$= y_1 \oplus y_3$
x_3	$= x_1 \oplus y_1$
	$= x_1 \oplus x_2 \oplus y_3$
	$= x_2 \oplus x_4 \oplus y_4$
	$= y_2 \oplus y_4$
x_4	$= x_2 \oplus y_2$
	$= x_2 \oplus x_3 \oplus y_4$

Table 5.2: Equations of decoding a custom XOR-(4,4) erasure code

Unlike MDS erasure codes, the XOR codes do not promise to tolerate any k fragments lost. As for the XOR-(4,4) code used here, it can tolerate any two fragments lost, but with potentially ability to tolerate three or four lost fragments depending on the loss pattern. Like many other XOR-based erasure codes, every set of the fragments are required to be enumerated for measuring the fault tolerance[39]. For example, if fragments $\{x_1, y_1, y_3, y_4\}$ are available, the remaining data fragments can be restored with them, and so as for the lost parity fragment. However consider another example of fragments $\{x_1, x_2, x_3, y_1\}$ are available, there is no chance to restore fragment x_4 as there is no redundancy for this data fragment.

The main advantage of using the XOR codes is the performance. Both encoding and decoding processes of XOR codes can be fulfilled with simple XOR operations, which can be done in sub-linear time. With such benefits, storage systems implemented with XOR-based erasure codes could reduce the computational overhead when encoding and decoding, while keeping the ability to provide extra data availability.

5.2.2 Implementation

The erasure code module in the system is implemented as a plugin in PyECLib, so that it can be utilized in Swift. As PyECLib is written in Python and has a highly extensible infrastructure so that it is possible to implement the XOR-(4,4) code mentioned in Section 5.2.1 as a custom backend of erasure code.

As for the erasure coding backends supported by PyECLib such as liberasurecode and Jerasure, a class `ECDriver` is used to manage and coordinate the third party libraries. For example, a RS-(4,2) erasure code module implemented by liberasurecode could be initialized as:

```
ec_driver = ECDriver(k = 4, m = 2, \
                ec_type = "liberasurecode_rs_vand")
```

Thus, to support the custom erasure code, 'ntnu_erasurecode' is defined as a new erasure code type, as well as class 'NTNU_EC_Driver' is defined to perform the functions. The following content of this section explains the three major functions of NTNU_EC_Driver: encode, decode and reconstruct. The entire source code of NTNU_EC_Driver could be found in Appendix A.

Encode

As the basis of the XOR-based erasure codes, the XOR operations are implemented with the Python's built-in cryptography toolkit Crypto library. As mentioned in [38], `strxor()` function in Crypto provided optimal performance. Listing 1 shows the modified function to perform XOR operations on two pieces of data with different length.

Listing 1 Source code of sxor() function for XOR operations

```
def sxor(s1, s2):
  s1_len = len(s1)
  s2_len = len(s2)
  if s1_len > s2_len:
    s2 = s2.ljust(s1_len,' ')
  elif s2_len > s1_len:
    s1 = s1.ljust(s2_len,' ')
  return Crypto.Util.strxor.strxor(s1,s2)
```

The implementation of encode function is quite straightforward, as listed in Listing 2. The function takes raw data as input, calculates the length of each fragment according to the parameter k of erasure code. After the data is divided into data fragments, the parity fragments are generated according to the encoding equations, with the use of `sxor()` function.

Listing 2 Source code of encode() function in NTNU_EC_Driver

```python
def encode(self, data_bytes):
  data = []; parity = []; output_fragments = []; offset = 0
  # Calculate fragment size and divide data
  fragment_size = int(math.ceil(len(data_bytes) / float(self.k)))
  for i in range(self.k - 1):
      fragment = data_bytes[offset : offset + fragment_size]
      data.append(fragment)
      offset += fragment_size
  # Append last fragment
  last_fragment = data_bytes[offset:]
  data.append(last_fragment)
  data_x1, data_x2, data_x3, data_x4 = data

  parity_y1 = sxor(data_x1, data_x3)
  parity_y2 = sxor(data_x2, data_x4)
  parity_y3 = sxor(sxor(data_x1, data_x2), data_x3)
  parity_y4 = sxor(sxor(data_x2, data_x3), data_x4)
  parity = [parity_y1, parity_y2, parity_y3, parity_y4]

  # Add indexes to fragments as metadata
  for i in range(len(data)):
      indexed_fragment = self._add_fragment_index(data[i], i)
      output_fragments.append(indexed_fragment)
  for i in range(len(parity)):
      indexed_fragment = self._add_fragment_index(parity[i],
↪  i+self.k)
      output_fragments.append(indexed_fragment)
  return output_fragments
```

The implementation of decode function is more complicated, as there could be several decoding equations for a single data fragments in XOR codes. With the ability to tolerate several fragments lost, the fragments passed to the function could be a subset of all fragments encoded. Thus, the idea of decoding is to restore as many missing data fragments from other data fragments and the redundant parity fragments as possible. Thus, each decoding equation for the missing data fragments needed to be inspected recursively, until there is no possible restoration. If all the data fragments are available after process, the data could be decoded and returned.

Otherwise an exception of insufficient fragments will be threw as the decoding is failed. Listing 3 shows the decode function, and Listing 4 shows the process of recursively attempting to restore the x_1 fragment.

Listing 3 Source code of decode() function in NTNU_EC_Driver

```
def decode(self, fragment_payloads, ranges=None,
               force_metadata_checks=False):
  if len(fragment_payloads) < self.k:
      raise ECInsufficientFragments( \
        "Insufficient fragments given to decode." )

  fragments = [None] * (self.k + self.m)
  all_data_index = set(range(self.k))
  available_fragment_index = set()

  # extract indexes from fragments
  for indexed_fragment in fragment_payloads:
      index, fragment = self._get_fragment_index(indexed_fragment)
      fragments[index] = fragment
      available_fragment_index.add(index)

  # recursively restore data fragments
  # until all data fragments are available
  # or no further restoration possible
  while(len(available_fragment_index & all_data_index) != self.k):
      pre_fragments_count = len(available_fragment_index)
      self._fix_data_fragment(fragments, available_fragment_index)
      post_fragments_count = len(available_fragment_index)
      if post_fragments_count == pre_fragments_count:
          raise ECInsufficientFragments( \
            "Insufficient fragments given to decode." )
          return

  # All data fragments are guaranteed to be repaired by now
  for index in all_data_index:
      assert fragments[index] is not None

  # decode original data
  data_bytes = ''
  for i in all_data_index:
      data_bytes += fragments[i]
  return data_bytes
```

Listing 4 Source code of __fix_data_fragment() function in NTNU_EC_Driver (partly)

```python
def _fix_data_fragment(self, all_fragments, available_fragment_index):
  all_data_index = set(range(self.k))
  missing_data_index = all_data_index - available_fragment_index

  if x1 in missing_data_index:
      if x3 in available_fragment_index \
        and y1 in available_fragment_index:
          all_fragments[x1] = sxor(all_fragments[x3], all_fragments[y1])
          available_fragment_index.add(x1)
      elif x2 in available_fragment_index \
        and x3 in available_fragment_index \
        and y3 in available_fragment_index:
          all_fragments[x1] = sxor(sxor(all_fragments[x2], all_fragments[x3]), all_fragments[y3])
          available_fragment_index.add(x1)

  # for x2 x3 x4 ...
```

The reconstruction process is quite similar to the decoding process. In addition to restoring all the data fragments with the recursive process mentioned previously, reconstruct function also attempts to generate the missing parity fragments, exactly as the encode function does.

Limitations

While the goal of the prototype is to implement and apply the XOR-(4,4) code to Swift, there is clearly some drawbacks and limitations of the implementation, due to the scope of the project.

As introduced previously, PyECLib calls backend libraries for performing erasure coding processes. Most of the backend libraries current available are written in C language. However, in this erasure code module prototype, the process is fully implemented with Python, as a plugin to PyECLib. Thus, there are inherently performance drawbacks for the prototype comparing to the C-written backends.

Additionally, the recursive decoding and reconstruction logics are not optimal, as there might be duplicate conditional judgments inside loops. One way to fix this problem is to hard code a bitmap for each data pattern and generate corresponding decoding procedures.

5.3 Results

As a prerequisite, the plugin of the implemented XOR-(4,4) code is compiled and integrated into PyECLib and Swift. The storage policy is defined and the corresponding ring is built with eight loopback devices. A container named 'ntnu-xor' is created with the storage policy specified. Moreover, a 200MB test file is generated as the data to backup.

To enable the interaction between Duplicity and Swift, the credential information of Swift should be set in the environment variable, which can be done as:

```
export SWIFT_USERNAME=test:tester
export SWIFT_PASSWORD=testing
export SWIFT_AUTHURL=http://127.0.0.1:8080/auth/v1.0
export SWIFT_AUTHVERSION=1
```

Duplicity is a command line based tool, so it can be triggered with simple commands in terminal. To generate a full backup of the target data, `full` option need to be specified in the command. As Duplicity supports to use Swift as backend repository, it uses `swift://` as suffix scheme to specify the remote destination as Swift, following with the container name. The encryption is enabled by default, thus a passphrase is requested to generate the encryption key. Figure 5.1 shows the execution process of backup using Duplicity.

Consequently, the original data is processed by Duplicity and sent to Swift for storage. All the generated files are demonstrated in Figure 5.2. Note that the files are not the fragments from erasure coding, but generated by Duplicity. The original data is sliced into eight parts, and two extra files are created as manifest and signature used for detecting incremental backup. Each file is then erasure coded by Swift and organized in the object ring.

The restore is similar to the reverse process of backup. Duplicity retrieves the decoded data from Swift, assembles the segments and then decrypts it with the same passphrase as used in encryption. Figure 5.3 shows the process of restoring and verifying a backup.

```
swift@swift-VirtualBox:~/duplicity_test$ duplicity full ./backup/ swift://ntnu-xor
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: none
GnuPG passphrase:
Retype passphrase to confirm:
--------------[ Backup Statistics ]--------------
StartTime 1464728109.06 (Wed Jun  1 04:55:09 2016)
EndTime 1464728122.59 (Wed Jun  1 04:55:22 2016)
ElapsedTime 13.54 (13.54 seconds)
SourceFiles 2
SourceFileSize 204804096 (195 MB)
NewFiles 2
NewFileSize 204804096 (195 MB)
DeletedFiles 0
ChangedFiles 0
ChangedFileSize 0 (0 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 2
RawDeltaSize 204800000 (195 MB)
TotalDestinationSizeChange 206073223 (197 MB)
Errors 0
-------------------------------------------------
```

Figure 5.1: Sample process of perform a full backup with Duplicity to Swift

5.4 System Analysis

A set of tests are performed to analyze the prototype of backup storage system. The implemented erasure code is compared with several other erasure coding schemes in terms of data durability and performance. The impact of encryption to the overall performance is also investigated in this section.

5.4.1 Test Setup

The tests are performed in the same SAIO virtual machine as introduced in Section 4.3.1. Detailed performance baseline is exactly the same as listed in Table 4.2. The version of Duplicity used in the tests is 0.6.23.

Together with the implemented XOR-(4,4) erasure code, alternative storage policies are also involved in the test, as listed in Table 5.3. The parameters of replica factor and erasure coding schemes are set to achieve similar data durability as the implemented XOR-(4,4) code provides. Note that there is (k, m) restriction for the flat_xor_hd_3 code. This test uses the scheme (5,5).

```
swift@swift-VirtualBox:~/duplicity_test$ swift -A http://127.0.0.1:8080/auth/v1.0 -U
 ↪  test:tester -K testing list --lh ntnu-xor
1.1M 2016-05-31 20:55:23 application/octet-stream
 ↪  duplicity-full-signatures.20160531T205500Z.sigtar.gpg
 470 2016-05-31 20:55:23 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.manifest.gpg
 25M 2016-05-31 20:55:10 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol1.difftar.gpg
 25M 2016-05-31 20:55:11 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol2.difftar.gpg
 25M 2016-05-31 20:55:13 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol3.difftar.gpg
 25M 2016-05-31 20:55:15 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol4.difftar.gpg
 25M 2016-05-31 20:55:17 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol5.difftar.gpg
 25M 2016-05-31 20:55:19 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol6.difftar.gpg
 25M 2016-05-31 20:55:21 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol7.difftar.gpg
 21M 2016-05-31 20:55:22 application/octet-stream
 ↪  duplicity-full.20160531T205500Z.vol8.difftar.gpg
197M
```

Figure 5.2: Sample Duplicity backup files stored in Swift

```
swift@swift-VirtualBox:~/duplicity_test$ duplicity restore swift://ntnu-xor
 ↪  ./restore; diff ./restore/backup.img ./backup/backup.img && echo Same || echo
 ↪  Different
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: Wed Jun  1 04:55:00 2016
GnuPG passphrase:
Same
```

Figure 5.3: Sample process of restore a full backup with Duplicity from Swift

5.4.2 Data Availability

As the implemented XOR based (4,4) code is not a MDS code, thus it is not optimal. It is guaranteed to tolerate any two missing fragments. However, in some cases it is also recoverable with three or four missing fragments, depending on the data loss pattern. With three fragments missing, the code is able to reconstruct the entire data in 52 out of 56 combinations of available fragments. With four fragments missing, the code is able to reconstruct the entire data in 45 out of 70 combinations of available fragments. The probability of a successful reconstruction for different count of available fragments are listed in Table 5.4.

To quantitatively compare the data availability provided by XOR-(4,4) code with

Storage Policy	**Description**
XOR-(4,4)	XOR-based erasure code implemented in this project.
3x Replication	Simple replication with three replicas.
RS-(6,3)	Reed-Solomon erasure encoding using Vandermonde matrices, implemented by liberasurecode.
XOR_HD_3-(5,5)	Flat-XOR based HD combination codes[14], implemented by liberasurecode.

Table 5.3: Storage polices used in the test as comparisons

Available fragments	Probability of successful reconstruction
$n = 6, 7, 8$	1.0
$n = 5$	$\frac{52}{56}$
$n = 4$	$\frac{45}{70}$
$n = 0, 1, 2, 3$	0.0

Table 5.4: Probability distribution of successful reconstruction for XOR-(4,4)

the competitors, similar approaches as listed in Section 4.2 are used. Assume that the availability of a single disk is 0.995 and the disk failure is independent. The overall availability is calculated based on Equation 4.2, together with the probability of successful reconstruction as the weight. The overall availability and the overhead of each storage policy are listed in Table 5.5. The results show that, the implemented XOR-(4,4) code is able to cut half of the storage overhead while keeping similar data durability when comparing to the 3x replication policy. However, due to the inherently designs, the XOR-based erasure codes are not as space efficiency as the Reed-Solomon codes.

Policy	**Guaranteed Failures Tolerated**	**Data Availability**	**Number of Nines**	**Space Overhead**
XOR-(4,4)	2	0.9999994969	6	100%
3x Replication	2	0.999999875	6	200%
RS-(6,3)	3	0.9999999228	7	50%
XOR_HD_3-(5,5)	2	0.9999861168	4	100%

Table 5.5: Availability and space overhead of storage policies used for Duplicity

5.4.3 Performance

The performance of test scenarios are tested by measuring the execution time for performing a full backup and restoration. The execution time of backup is collected from the backup statistics generated by Duplicity, and bash command `time` is used to measure the time for restoration. In order to avoid the interrupt of requesting passphrase for encryption, the passphrase is pre-defined in the system environment variables. Each iteration of test is forced to set as a full backup, targeting an empty container with corresponding storage policy in Swift.

Performance of encryption

Duplicity uses symmetric encryption by default, which provides basic data protection at a relatively high speed. Alternatively, Duplicity also supports to use asymmetric encryption or generate backups without encryption. As the encryption is involved in the backup process, there might be extra computational overhead. Three tests are performed to test the performance impact, with symmetric encryption, asymmetric encryption and no encryption respectively. As for the asymmetric encryption, a 2048 bit long RSA-RSA key is generated with GnuPG for the use of encryption in Duplicity (no sign is involved). No replication or erasure coding mechanism is enabled in this test in order to avoid disturbance. The test data in use is a randomly generated 200MB file.

The execution time of backup and restoration for different test scenarios are demonstrated in Figure 5.4. The results show that enabling encryption does have extra computational cost, especially for the decrypt process when restoring the backups. Moreover, it is worth noting that there is hardly any performance gap between symmetric and asymmetric encryptions in Duplicity.

Performance of erasure codes

A set of tests are performed to evaluate the performance of the implemented XOR-(4,4) code, comparing to the competing schemes listed previously. A 1GB randomly generated file is used as the test data, in order to simulate a backup image in real scenarios, which is generally a single large file. All the tests are performed with encryption disabled. The execution time of both backup and restoration are recorded, and presented in Figure 5.5.

The results show that the backup time are almost the same for each storage policy. One explanation of that is that Duplicity segments the original file before handing to Swift. As listed previously, the data is sliced into a bunch of small compressed file, with size of 25MB. Thus, the backup of a large single file turns to writing a series of small files.

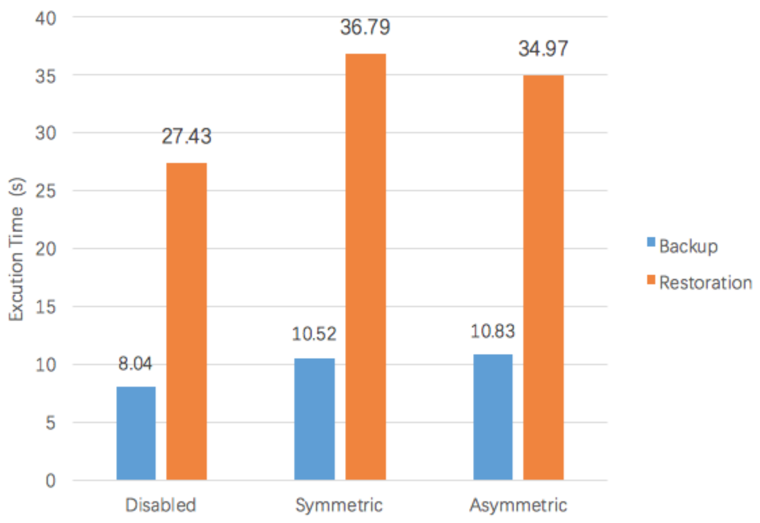

Figure 5.4: Execution time of Duplicity using different encryption options

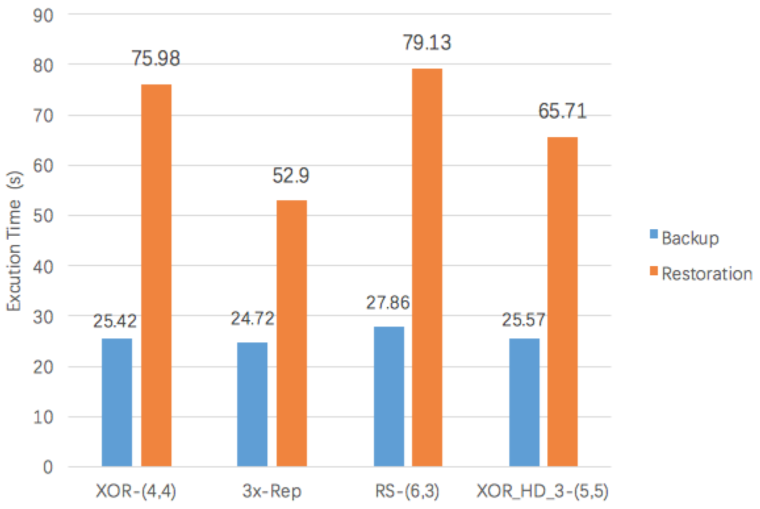

Figure 5.5: Execution time of Duplicity using different storage policies

As for the restoration process, the simple replication approach has the best performance, which is reasonable since there is no extra computational cost required for decoding. Both XOR-based erasure codes outperforms the Reed-Solomon code, which conforms to the theory that simple XOR operations save more time than the multiply operations in Galois field. Between two XOR codes, there is also an obvious difference in restoration time. This might be caused by unoptimized Python-based implementation of XOR-(4,4) code, which is discussed in Section 5.2.2.

Another factor that may have disturbance on the performance results is that, the incomplete support for the XOR-based erasure codes in Swift. In order to response as fast as possible, Swift uses the mechanism to call the erasure code library with a certain number (equal to k in most cases) of fragments, which are the first arriving at the proxy server, while the others are discarded. This works for optimal MDS codes. However, non-optimal codes like XOR based codes do not promise to reconstruct from m fragments missing. Instead, in some cases the original data is not possible to be recovered. Therefore, more attempts are made, until the decoder has sufficient fragments for decoding. This defect causes potentially extended decoding time for non-optimal codes, which certainly affects the performance.

5.4.4 Discussion

The results of the analysis demonstrated the quality of the custom XOR erasure code implemented. The mathematical analysis proved that it is able to provide similar data durability as comparing to triple replication, while significantly reduce the storage overhead at the same time. The results of real tests also showed the performance promising towards Reed-Solomon codes, although the gap is not that big. However, consulting the C-written flat_xor_hd code implemented by liberasurecode, the proposed XOR-(4,4) code could be expected to gain a performance improvement after proper optimization.

Chapter 6 Conclusions

This project investigated the erasure coding technique, with the usage in OpenStack Swift.

Erasure coding is a novel approach applied in storage systems in order to provide additional data reliability. The main advantage of erasure coding is the improved data availability with relatively low extra cost on the storage space. There are different types of erasure codes, with different implementations. Thus, erasure codes are highly flexible as it can be adjusted with different schemes to meet the variety of requirements including data availability, performance and storage space. With such features, erasure coding can be considered as an alternative choice to the replication approach, and applied in real use cases.

The report demonstrated different stages of fulfilling the objectives listed in Chapter 1:

– The background information of the project was explained in Chapter 2. A study was performed for two widely-used approaches of enhancing data availability: replication and RAID. The pros and cons of both approaches were analyzed. Erasure coding was then explained with the concepts and designs behind. Several erasure code families were also introduced in this chapter.

– Chapter 3 introduced OpenStack Swift, which is another focus of this project. The basic concepts and fundamental infrastructures were firstly introduced. Two tutorials showed the configuration of Swift, and the integration of erasure coding to Swift.

– Chapter 4 performed a system comparison between replication and erasure coding. The data durability and overhead provided by both approaches were calculated mathematically. A series of tests analyzed the performance of both systems in different aspects.

 – An implementation of a custom erasure code scheme is presented in Chapter 5. It was used together with Duplicity to build a prototype of backup storage system as a real scenario, followed with analysis of the system.

Discussion and Future Works

The project presents erasure coding in different areas both theoretically and practically. The outcome of the project could prove the feasibility of applying erasure codes in OpenStack Swift and other storage systems. Meanwhile, the implemented erasure code module could also be used in further investigation and development. As shown in the analysis, the erasure coding is suitable in certain use cases. Although it does have limitations on the computational cost, erasure coding comes as a tradeoff to provide balance between the performance and price. As concluded in the report, erasure codes suit the environments where the storage capacity is sensitive with huge amount of rarely-accessed data.

However, there are some limitations of the project. First of all, this project focused on the common Reed-Solomon codes and the XOR-based code implemented. Yet some other types of erasure codes with different designs are also studied and used, such as the Local Reconstruction Codes (LRCs) used by Microsoft[2]. Besides, due to the limited scope, the tests in this project were performed on single host. There might be other factors affecting the results in a multi-node cluster environment.

There could be several works as the follow-up of this project. One thing is to rewrite the prototype of erasure code module in C language, in order to achieve better performance. Meanwhile, it is also interesting to study the feasibility and implementation of a storage system with the hybrid of both replication and erasure codes, so that it could have rapid read performance through replicas and benefit from erasure codes for serious data loss.

References

[1] marketsandmarkets.com, “Cloud storage market by solutions (primary storage solution, backup storage solution, cloud storage gateway solution, and data movement and access solution) - global forecast and analysis to 2020,” August 2015. Last Accessed: May 2016.

[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure coding in windows azure storage,” in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pp. 15–26, 2012.

[3] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “Xoring elephants: Novel erasure codes for big data,” in *Proceedings of the VLDB Endowment*, vol. 6, pp. 325–336, VLDB Endowment, 2013.

[4] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in globally distributed storage systems.,” in *OSDI*, pp. 61–74, 2010.

[5] D. Borthakur, “Hdfs architecture guide.” https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Data+Replication. Last Accessed: May 2016.

[6] D. A. Patterson, G. Gibson, and R. H. Katz, *A case for redundant arrays of inexpensive disks (RAID)*, vol. 17. ACM, 1988.

[7] SNIA, “Common raid disk data format (ddf).” http://www.snia.org/tech_activities/standards/curr_standards/ddf. Last Accessed: May 2016.

[8] J. Nonnenmacher, E. W. Biersack, and D. Towsley, “Parity-based loss recovery for reliable multicast transmission,” *Networking, IEEE/ACM Transactions on*, vol. 6, no. 4, pp. 349–361, 1998.

[9] A. Fujimura, S. Y. Oh, and M. Gerla, “Network coding vs. erasure coding: Reliable multicast in ad hoc networks,” in *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pp. 1–7, IEEE, 2008.

[10] N. Bhatnagar, K. M. Greenan, R. Wacha, E. L. Miller, and D. D. Long, "Energy-reliability tradeoffs in sensor network storage," in *In Proceedings of the 5th Workshop on Embedded Networked Sensors*, Citeseer, 2008.

[11] N. He, J. Cao, Z. Li, and Y. Ren, "Jvec: Joint video adaptation and erasure code for wireless video streaming broadcast," in *Communications (ICC), 2010 IEEE International Conference on*, pp. 1–5, IEEE, 2010.

[12] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM SIGCOMM computer communication review*, vol. 27, no. 2, pp. 24–36, 1997.

[13] K. M. Greenan, D. D. Long, E. L. Miller, T. J. Schwarz, and J. J. Wylie, "A spin-up saved is energy earned: Achieving power-efficient, erasure-coded storage.," in *HotDep*, 2008.

[14] K. M. Greenan, X. Li, and J. J. Wylie, "Flat xor-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–14, IEEE, 2010.

[15] A. G. Dimakis, P. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *Information Theory, IEEE Transactions on*, vol. 56, no. 9, pp. 4539–4551, 2010.

[16] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[17] Y. Fu, J. Shu, and Z. Shen, "Ec-frm: An erasure coding framework to speed up reads for erasure coded cloud storage systems," in *Parallel Processing (ICPP), 2015 44th International Conference on*, pp. 480–489, IEEE, 2015.

[18] J. S. Plank, "Erasure codes for storage systems: A brief primer," *Usenix magazine*, vol. 38, December 2013.

[19] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, *et al.*, "A performance evaluation and examination of open-source erasure coding libraries for storage.," in *FAST*, vol. 9, pp. 253–265, 2009.

[20] K. M. Greenan, E. L. Miller, and J. J. Wylie, "Reliability of flat xor-based erasure codes on heterogeneous devices," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 147–156, IEEE, 2008.

[21] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," 1995.

[22] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Peer-to-Peer Systems*, pp. 328–337, Springer, 2002.

[23] OpenStack, "Openstack open source cloud computing software." http://www.openstack.org/. Last Accessed: May 2016.

[24] OpenStack, “Openstack swift documentation.” http://swift.openstack.org. Last Accessed: May 2016.

[25] OpenStack, “Openstack administration guide - object storage.” http://docs.openstack.org/admin-guide/objectstorage.html. Last Accessed: May 2016.

[26] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, “Object storage: The future building block for storage systems,” in *Local to Global Data Interoperability-Challenges and Technologies, 2005*, pp. 119–123, IEEE, 2005.

[27] M. Mesnier, G. R. Ganger, and E. Riedel, “Object-based storage,” *Communications Magazine, IEEE*, vol. 41, no. 8, pp. 84–90, 2003.

[28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web,” in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 654–663, ACM, 1997.

[29] J. Dickinson, “Erasure codes now a part of openstack kilo.” https://swiftstack.com/blog/2015/04/30/erasure-codes-now-a-part-of-openstack-kilo/. Last Accessed: May 2016.

[30] J. S. Plank and K. M. Greenan, “Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0,” tech. rep., Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[31] T. L. (Intel), “Intel® intelligent storage acceleration library performance under xen* project hypervisor.” https://software.intel.com/en-us/articles/intel-isa-l-library-performance-under-xen-project-hypervisor. Last Accessed: May 2016.

[32] OpenStack, “Saio - swift all in one.” http://docs.openstack.org/developer/swift/development_saio.html. Last Accessed: May 2016.

[33] S. Inc., “ssbench - benchmarking tool for swift clusters.” https://github.com/swiftstack/ssbench. Last Accessed: May 2016.

[34] B. Escoto, K. Loafman, *et al.*, “Duplicity.” http://duplicity.nongnu.org/. Last Accessed: May 2016.

[35] A. Tridgell, P. Mackerras, *et al.*, “Rsync - an utility that provides fast incremental file transfer..” https://rsync.samba.org. Last Accessed: May 2016.

[36] W. Koch *et al.*, “The GNU privacy guard.” https://gnupg.org. Last Accessed: May 2016.

[37] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, “OpenPGP Message Format,” RFC 4880, RFC Editor, November 2007. http://www.rfc-editor.org/rfc/rfc4880.txt.

[38] H. N. Matland, “Intelligent scheduled backup using duplicity,” 2015.

[39] J. J. Wylie and R. Swaminathan, “Determining fault tolerance of xor-based erasure codes efficiently,” in *Dependable Systems and Networks, 2007. DSN’07. 37th Annual IEEE/IFIP International Conference on*, pp. 206–215, IEEE, 2007.

Appendix A
Source code of NTNU_EC_Driver class

```
import Crypto.Util.strxor

x1 = 0
x2 = 1
x3 = 2
x4 = 3
y1 = 4
y2 = 5
y3 = 6
y4 = 7

def sxor(s1, s2):
  s1_len = len(s1)
  s2_len = len(s2)
  if s1_len > s2_len:
    s2 = s2.ljust(s1_len,' ')
  elif s2_len > s1_len:
    s1 = s1.ljust(s2_len,' ')
  return Crypto.Util.strxor.strxor(s1,s2)

class NTNU_EC_Driver(object):

    def __init__(self, k, m, hd, ec_type=None, chksum_type=None,
                 validate=False):
        with open("/home/swift/log.txt", "a") as fpp:
            fpp.write("-init\n")
            fpp.close()

        if ec_type != PyECLib_EC_Types.get_by_name("ntnu_erasurecode"):
            raise ECBackendNotSupported(
                        "%s is not a supported type." % ec_type)

        if k != 4 or m != 4:
            raise ECBackendNotSupported(
                        "NTNU code only supports scheme (4, 4). (%d, %d) is given." % (k, m))

        if chksum_type != None:
            raise ECBackendNotSupported(
                        "No checksum implemented. Should set to None.")

        self.k = k
        self.m = m
        self.ec_type = ec_type

        # hd & validata ignored
        # self.hd = hd
        # self.validate = validate

    def _add_fragment_index(self, fragment, index):
        # ---------------------------------------------------------
        # WORKAROUND
        # Add index in front of each fragment, as metadata
        # with format: "%d,%s" % (index, data)
        # for (4,4) ntnu code the length of index is 1 (one-digit)
        return "%d,%s" % (index, fragment)

    def _get_fragment_index(self, indexed_fragment):
        # ---------------------------------------------------------
        # WORKAROUND
        # Extract index from the indexed fragment
        # return a tuple: (index, fragment)
        separator = indexed_fragment.index(",")
        index = int(indexed_fragment[:separator])
        fragment = indexed_fragment[separator+1:]
        return (index, fragment)

    def _fix_data_fragment(self, all_fragments, available_fragment_index):
        all_data_index = set(range(self.k))
        missing_data_index = all_data_index - available_fragment_index

        if x1 in missing_data_index:
            if x3 in available_fragment_index and y1 in available_fragment_index:
                all_fragments[x1] = sxor(all_fragments[x3], all_fragments[y1])
                available_fragment_index.add(x1)
            elif x2 in available_fragment_index and x3 in available_fragment_index and y3 in available_fragment_index:
                all_fragments[x1] = sxor(sxor(all_fragments[x2], all_fragments[x3]), all_fragments[y3])
                available_fragment_index.add(x1)
        if x2 in missing_data_index:
            if x4 in available_fragment_index and y2 in available_fragment_index:
                all_fragments[x2] = sxor(all_fragments[x4], all_fragments[y2])
                available_fragment_index.add(x2)
            elif x1 in available_fragment_index and x3 in available_fragment_index and y3 in available_fragment_index:
                all_fragments[x2] = sxor(sxor(all_fragments[x1], all_fragments[x3]), all_fragments[y3])
                available_fragment_index.add(x2)
            elif x3 in available_fragment_index and x4 in available_fragment_index and y4 in available_fragment_index:
                all_fragments[x2] = sxor(sxor(all_fragments[x3], all_fragments[x4]), all_fragments[y4])
                available_fragment_index.add(x2)
            elif y1 in available_fragment_index and y3 in available_fragment_index:
```

```python
                all_fragments[x2] = sxor(all_fragments[y1],
↪   all_fragments[y3])
                available_fragment_index.add(x2)
        if x3 in missing_data_index:
            if x1 in available_fragment_index and y1 in
↪   available_fragment_index:
                all_fragments[x3] = sxor(all_fragments[x1],
↪   all_fragments[y1])
                available_fragment_index.add(x3)
            elif x1 in available_fragment_index and x2 in
↪   available_fragment_index and y3 in available_fragment_index:
                all_fragments[x3] = sxor(sxor(all_fragments[x1],
↪   all_fragments[x2]), all_fragments[y3])
                available_fragment_index.add(x3)
            elif x2 in available_fragment_index and x4 in
↪   available_fragment_index and y4 in available_fragment_index:
                all_fragments[x3] = sxor(sxor(all_fragments[x2],
↪   all_fragments[x4]), all_fragments[y4])
                available_fragment_index.add(x3)
            elif y2 in available_fragment_index and y4 in
↪   available_fragment_index:
                all_fragments[x3] = sxor(all_fragments[y2],
↪   all_fragments[y4])
                available_fragment_index.add(x3)
        if x4 in missing_data_index:
            if x2 in available_fragment_index and y2 in
↪   available_fragment_index:
                all_fragments[x4] = sxor(all_fragments[x2],
↪   all_fragments[y2])
                available_fragment_index.add(x4)
            elif x2 in available_fragment_index and x3 in
↪   available_fragment_index and y4 in available_fragment_index:
                all_fragments[x4] = sxor(sxor(all_fragments[x2],
↪   all_fragments[x3]), all_fragments[y4])
                available_fragment_index.add(x4)


    def _fix_parity_fragment(self, all_fragments,
↪   available_fragment_index):
        all_parity_index = set(range(self.k, self.k + self.m))
        missing_parity_index = all_parity_index - available_fragment_index

        if y1 in missing_parity_index:
            all_fragments[y1] = sxor(all_fragments[x1], all_fragments[x3])
            available_fragment_index.add(y1)
        if y2 in missing_parity_index:
            all_fragments[y2] = sxor(all_fragments[x2], all_fragments[x4])
            available_fragment_index.add(y2)
        if y3 in missing_parity_index:
            all_fragments[y3] = sxor(sxor(all_fragments[x1], all_fragments[x2]), all_fragments[x3])
            available_fragment_index.add(y3)
        if y4 in missing_parity_index:
            all_fragments[y4] = sxor(sxor(all_fragments[x2], all_fragments[x3]), all_fragments[x4])
            available_fragment_index.add(y4)


    def encode(self, data_bytes):
        """
        Encode N bytes of a data object into k (data) + m (parity) fragments::

        def encode(self, data_bytes)

        input:   data_bytes - input data object (bytes)
        returns: list of fragments (bytes)
        throws:
          ECBackendInstanceNotAvailable - if the backend library cannot be found
          ECBackendNotSupported - if the backend is not supported by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were provided
          ECOutOfMemory - if the process has run out of memory
          ECDriverError - if an unknown error occurs
        """

        # Main fragment size
        fragment_size = int(math.ceil(len(data_bytes) / float(self.k)))

        data = []
        parity = []
        output_fragments = []

        offset = 0

        for i in range(self.k - 1):
            fragment = data_bytes[offset : offset + fragment_size]
            data.append(fragment)
            offset += fragment_size

        # Append last fragment
        last_fragment = data_bytes[offset:]
        data.append(last_fragment)

        data_x1, data_x2, data_x3, data_x4 = data

        parity_y1 = sxor(data_x1, data_x3)
        parity_y2 = sxor(data_x2, data_x4)

        parity_y3 = sxor(sxor(data_x1, data_x2), data_x3)
        parity_y4 = sxor(sxor(data_x2, data_x3), data_x4)

        parity = [parity_y1, parity_y2, parity_y3, parity_y4]

        # Add indexes to fragments as metadata
        for i in range(len(data)):
            indexed_fragment = self._add_fragment_index(data[i], i)
            output_fragments.append(indexed_fragment)

        for i in range(len(parity)):
            indexed_fragment = self._add_fragment_index(parity[i], i+self.k)
            output_fragments.append(indexed_fragment)

        return output_fragments


    def decode(self, fragment_payloads, ranges=None,
               force_metadata_checks=False):
        """
        Decode between k and k+m fragments into original object::

        def decode(self, fragment_payloads)

        input:   list of fragment_payloads (bytes)
        returns: decoded object (bytes)
        throws:
          ECBackendInstanceNotAvailable - if the backend library cannot be found
          ECBackendNotSupported - if the backend is not supported by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were provided
          ECOutOfMemory - if the process has run out of memory
          ECInsufficientFragments - if an insufficient set of fragments has been provided (e.g. not enough)
          ECInvalidFragmentMetadata - if the fragment headers appear to be corrupted
          ECDriverError - if an unknown error occurs
        """
        if ranges is not None:
            raise ECDriverError("Decode does not support range requests in the NTNU_EC_Driver.")

        if len(fragment_payloads) < self.k:
            raise ECInsufficientFragments(
                "Insufficient fragments given to decode.")

        fragments = [None] * (self.k + self.m)
        available_fragment_index = set()
        # extract indexes from fragments
        for indexed_fragment in fragment_payloads:
            index, fragment = self._get_fragment_index(indexed_fragment)
            fragments[index] = fragment
            available_fragment_index.add(index)


        all_data_index = set(range(self.k))

        # recursively restore data fragments
        # until all data fragments are available
        # or no further restoration possible
        while(len(available_fragment_index & all_data_index) != self.k):

            pre_fragments_count = len(available_fragment_index)

            self._fix_data_fragment(fragments, available_fragment_index)

            post_fragments_count = len(available_fragment_index)

            if post_fragments_count == pre_fragments_count:
                raise ECInsufficientFragments("Insufficient fragments given to decode.")
                return

        # All data fragments are guaranteed to be repaired by now
        for index in all_data_index:
            assert fragments[index] is not None

        # decode original data
        data_bytes = ''

        for i in all_data_index:
            data_bytes += fragments[i]

        return data_bytes
```

```
    def reconstruct(self, available_fragment_payloads,
                    missing_fragment_indexes):
        """
        Reconstruct "missing_fragment_indexes" using "available_fragment_payloads"::

        def reconstruct(self, available_fragment_payloads, missing_fragment_indexes)

        input: available_fragment_payloads - list of fragment payloads
        input: missing_fragment_indexes - list of indexes to reconstruct
        output: list of reconstructed fragments corresponding to missing_fragment_indexes
        throws:
          ECBackendInstanceNotAvailable - if the backend library cannot be found
          ECBackendNotSupported - if the backend is not supported by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were provided
          ECOutOfMemory - if the process has run out of memory
          ECInsufficientFragments - if an insufficient set of fragments has been provided (e.g. not enough)
          ECInvalidFragmentMetadata - if the fragment headers appear to be corrupted
          ECDriverError - if an unknown error occurs
        """
        with open("/home/swift/log.txt", "a") as fpp:
            fpp.write("-reconstruct\n")
            fpp.close()

        fragments = [None] * (self.k + self.m)
        available_fragment_index = set()

        for indexed_fragment in available_fragment_payloads:
            index, fragment = self._get_fragment_index(indexed_fragment)
            fragments[index] = fragment
            available_fragment_index.add(index)

        all_data_index = set(range(self.k))
        all_parity_index = set(range(self.k, self.k + self.m))

        while(len(available_fragment_index & all_data_index) !=
↪   self.k):

            pre_fragments_count = len(available_fragment_index)

            self._fix_data_fragment(fragments,
↪   available_fragment_index)

            post_fragments_count = len(available_fragment_index)

            if post_fragments_count == pre_fragments_count:
                raise ECInsufficientFragments("Insufficient
↪   fragments given to decode.")
                return

        # All data fragments are repaired by now
        for index in all_data_index:
            assert fragments[index] is not None

        self._fix_parity_fragment(fragments,
↪   available_fragment_index)

        # All parity fragments are repaired by now
        for index in all_parity_index:
            assert fragments[index] is not None


        return fragments

    def fragments_needed(self, missing_fragment_indexes):
        """
        Return the indexes of fragments needed to reconstruct
↪   "missing_fragment_indexes"::

        def fragments_needed(self, missing_fragment_indexes)

        input: list of missing_fragment_indexes
        output: list of fragments needed to reconstruct fragments
    ↪ listed in missing_fragment_indexes
        throws:
          ECBackendInstanceNotAvailable - if the backend library
    ↪ cannot be found
          ECBackendNotSupported - if the backend is not supported
    ↪ by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were
    ↪ provided
          ECOutOfMemory - if the process has run out of memory
          ECDriverError - if an unknown error occurs
        """
        with open("/home/swift/log.txt", "a") as fpp:
            fpp.write("-needed\n")
            fpp.close()

        all_fragments = range(self.k + self.m)

        return list(set(all_fragments) -
    ↪ set(missing_fragment_indexes))


    def min_parity_fragments_needed(self):
        """
        Minimum parity fragments needed for durability
    ↪ gurantees::

        def min_parity_fragments_needed(self)

        #  NOTE: Currently hard-coded to 1, so this can only be
    ↪ trusted for MDS codes, such as Reed-Solomon.

        output: minimum number of additional fragments needed to
    ↪ be synchronously written to tolerate the loss of any one
    ↪ fragment (similar guarantees to 2 out of 3 with 3x
    ↪ replication)
        throws:
          ECBackendInstanceNotAvailable - if the backend library cannot be found
          ECBackendNotSupported - if the backend is not supported by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were provided
          ECOutOfMemory - if the process has run out of memory
          ECDriverError - if an unknown error occurs
        """
        with open("/home/swift/log.txt", "a") as fpp:
            fpp.write("-min parity\n")
            fpp.close()

        # Hard-coded to 1, as PyECLibDriver (for RS codes & XOR codes)
        return 1

    def get_metadata(self, fragment, formatted=0):
        """
        Return an opaque header known by the underlying library or a formatted header (Python dict)::

        def get_metadata(self, fragment, formatted = 0)

        input: raw fragment payload
        input: boolean specifying if returned header is opaque buffer or formatted string
        output: fragment header (opaque or formatted)
        throws:
          ECBackendInstanceNotAvailable - if the backend library cannot be found
          ECBackendNotSupported - if the backend is not supported by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were provided
          ECOutOfMemory - if the process has run out of memory
          ECDriverError - if an unknown error occurs
        """
        with open("/home/swift/log.txt", "a") as fpp:
            fpp.write("-get meta\n")
```

```python
            fpp.close()
        return ''

    def verify_stripe_metadata(self, fragment_metadata_list):
        """
        Use opaque buffers from get_metadata() to verify a the consistency of a stripe::

        def verify_stripe_metadata(self, fragment_metadata_list)

        intput: list of opaque fragment headers
        output: formatted string containing the 'status' (0 is success) and 'reason' if verification fails
        throws:
          ECBackendInstanceNotAvailable - if the backend library cannot be found
          ECBackendNotSupported - if the backend is not supported by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were provided
          ECOutOfMemory - if the process has run out of memory
          ECDriverError - if an unknown error occurs
        """
        with open("/home/swift/log.txt", "a") as fpp:
            fpp.write("-verifystripe\n")
            fpp.close()
        return True

    def get_segment_info(self, data_len, segment_size):
        """
        Return a dict with the keys - segment_size, last_segment_size, fragment_size, last_fragment_size and num_segments::

        def get_segment_info(self, data_len, segment_size)

        input: total data_len of the object to store
        input: target segment size used to segment the object into multiple EC stripes
        output: a dict with keys - segment_size,
↪   last_segment_size, fragment_size, last_fragment_size and
↪   num_segments
        throws:
          ECBackendInstanceNotAvailable - if the backend library
↪   cannot be found
          ECBackendNotSupported - if the backend is not supported
↪   by PyECLib (see ec_types above)
          ECInvalidParameter - if invalid parameters were
↪   provided
          ECOutOfMemory - if the process has run out of memory
          ECDriverError - if an unknown error occurs

        """
        with open("/home/swift/log.txt", "a") as fpp:
            fpp.write("-getsegment\n")
            fpp.close()



        num_segments = int(math.ceil(float(data_len) /
↪   segment_size))

        fragment_size = int(math.ceil(float(data_len) / self.k))
        last_fragment_size = data_len - fragment_size * (self.k -
↪   1)

        if num_segments == 1:
            segment_size = data_len
            last_segment_size = data_len
        else:
            last_segment_size = data_len - segment_size *
↪   (num_segments - 1)

        #
↪   ---------------------------------------------------------
        # WORKAROUND
        # Add index in front of each fragment, as metadata
        # with format: (%d,%s) % (index, data)
        # for (4,4) ntnu code the length of index is 1 (one-digit)
        index_overhead = 2
        fragment_size += index_overhead
        last_fragment_size += index_overhead
        # ---------------------------------------------------------

        results = {
            "segment_size": segment_size,
            "last_segment_size": last_segment_size,
            "fragment_size": fragment_size,
            "last_fragment_size": last_fragment_size,
            "num_segments": num_segments
        }

        return results
```